Discovering hidden dependencies in constraint-based declarative process models for improving understandability

Discovering Hidden Dependencies in Constraint-Based Declarative Process Models for Improving Understandability

Johannes De Smedt[a,b,*], Jochen De Weerdt[a], Estefanía Serral[a], Jan Vanthienen[a]

[a] *KU Leuven Faculty of Economics and Business, Department of Decision Sciences and Information Management*
[b] *University of Edinburgh Business School, Management Science and Business Economics Group*

Abstract

Flexible systems and services require a solid approach for modeling and enacting dynamic behavior. Declarative process models gained plenty of traction lately as they have proven to provide a good fit for the problem at hand, i.e. visualizing and executing flexible business processes. These models are based on constraints that impose behavioral restrictions on process behavior. Essentially, a declarative model is a set of constraints defined over the set of activities in a process. While allowing for very flexible process specifications, a major downside is that the combination of constraints can lead to behavioral restrictions not explicitly visible when reading a model. These restrictions, so-called hidden dependencies, make the models much more difficult to understand. This paper presents a technique for discovering hidden dependencies and making them explicit by means of dependency structures. Experiments with novice process modelers demonstrate that the proposed technique lowers the cognitive effort necessary to comprehend a constraint-based process model.

Keywords: Declarative process modeling, Declare, Hidden dependencies, Constraint-based process models, Model comprehension, Empirical research.

*Corresponding author
Email address: Johannes.DeSmedt@{kuleuven.be;ed.ac.uk} (Johannes De Smedt)

1. Introduction

Declarative process models (DPMs) have been proposed to counter the limitations of procedural modeling languages to represent and execute processes flexibly [1, 2], a desirable feature that is of importance in application areas such as healthcare [3] or services [4]. Instead of modeling predetermined paths of activities, declarative process models typically use constraints or rules to express what can, cannot, and must happen. Every execution sequence that is not strictly forbidden by the constraints can be enacted by the model, which makes it very flexible. Many executions are possible due to the interaction of the constraints over the activities. However, each type of constraint can have distinctive effects on the enabledness of an activity, creating dependencies that are not explicit or visible in the graphical model or even in the execution semantics. The dependencies between constraints and activities that are not explicit or visible in the model [5, 6, 7] are so-called hidden dependencies and make declarative models difficult to comprehend [6, 8].

This paper proposes a technique capable of revealing hidden dependencies in constraint-based declarative process models by propagating the constraints' properties through the activities of a model and builds upon the prior work in [9]. It extends this work by explicitly addressing how constraints propagate their restrictions over activities, as illustrated in Section 3, and forms the backdrop for building dependency structures for the whole model. They then are used to visualize the dependencies, as well as create textual annotations. In this way, the paper addresses two suggestions for improvement that were found in the user study performed in [6], i.e. 'Simplify combination on constraints' by introducing an extra layer of annotation that explains their relations, and 'Make hidden dependencies explicit' by capturing them in a visual model and textual annotations. The technique has been implemented for the Declare language [10], one of the most-widely used declarative process languages, and is implemented in

the newly-developed Declare Execution Environment[1], a tool that supplements an existing Declare model with visual and textual annotations and shows which behavior is allowed or disallowed by the model and why.

The technique has been tested in an empirical evaluation in which 146 novice modelers participated. The results are reported in greater detail with a more in-depth statistical analysis using more factors, and a bigger sample than in [9]. The evaluation shows that hidden dependencies pose a significant burden for the modelers to understand the full behavior of a model, and demonstrates that our technique actually has an impact on the cognitive complexity and improves the understandability of declarative process models by uncovering these dependencies. Using the proposed technique, modelers were better capable of getting a holistic view on the model they had to interpret, and were better capable of answering questions regarding the behavior of the model correctly.

The structure of the paper is as follows. First, in Section 2, the concept and background of DPMs are summarized and relevant characteristics are explained. In Section 3, the concept of a constraint-based DPM and the constraint propagations in Declare are formalized. In Section 4, the construction of constraint dependency structures is elaborated on. In Section 5, the tool is introduced which we developed to implement the ideas. Section 6 reports on the results of an experiment performed on users of the tool, and Section 7 concludes the paper with an overview of the results and future work.

2. State-of-the-art

In this section, the state-of-the-art of declarative process models and the research on their understandability is discussed.

2.1. Languages and Approaches

There exist numerous types of declarative process modeling languages in literature. The term surfaced with the proposal of EMBrA^{2}CE [2], DecSer-

[1] `http://www.processmining.be/declareexecutionenvironment`

Flow, and Condec [11, 1]. The former proposes an extension of Semantics and Business Rules Vocabulary [12] to model business processes, while the latter proposed a framework grounded in Linear Temporal Logic (LTL)-based constraints and later was transformed into the Declare language [10]. A similar approach was followed for DCR Graphs [13], a framework also based on a set of constraints, which is smaller but when used in a composite way offers at least the same expressiveness as LTL [14]. These contributions focus mainly on the control flow, though data-aware extensions exist, e.g., [15]. Other research focuses on a declarative specification of artifacts, including Guard Stage Milestone [16]. In recent years, however, Declare has seen the biggest surge in terms of research interest in many application areas such as process mining [17], application development [18], and process verification [19]. Therefore, it will be used to illustrate the example in this work. Other languages also support Declare for model checking as well, i.e. in SCIFF [4]. As a framework, it aims to support different semantics besides LTL, such as regular expressions [20, 21], and R/I-nets [22]. Furthermore, the template base is extensible, allowing Declare to support any constraint that can be defined with the same semantics. An overview of the Declare semantics in LTL and regular expressions are given in Table 1.

Declare uses a graphical notation that depicts activities as boxes and constraints as edges of various types. All such edges are annotated with their respective name, except for unary activities for they are either similar to cardinalities, or contain the constraint name. A major downside of the framework however, is that the graphical notation and execution model are separated. This downside was overcome in DCR Graphs by representing the states explicitly in the form of markings.

2.2. Understandability of Declarative Process Models and Hidden Dependencies

The departure of defining explicit control flow as in Business Process Model and Notation [23] or Petri nets [24], has urged a number of researchers to study the understandability and usability of DPMs in general. In [8, 25] , the com-

	Template	LTL Formula [1]	Regular Expression [21]	Description
Unaries	Existence(a,n)	$\Diamond(a \wedge \bigcirc(existence(n-1,a)))$	.*(a.*){n}	Activity a happens at least n times.
	Absence(a,n)	$\neg existence(n,a)$	[^a]*(a?[^a]*){n-1}	Activity a happens at most n times.
	Exactly(a,n)	$existence(n,a) \wedge absence(n+1,a)$	[^a]*(a[^a]*){n}	Activity a happens exactly n times.
	Init(a)	a	(a.*)?	Each instance has to start with activity a.
	Last(a)	$\Box(a \implies \neg X \neg a)$	.*a	Each instance has to end with activity a.
Unordered	Responded existence(a,b)	$\Diamond a \implies \Diamond b$	[^a]*((a.*b.*) |(b.*a.*))?	If a happens at least once then b has to happen or happened before a.
	Co-existence(a,b)	$\Diamond a \Longleftarrow \Diamond b$	[^ab]*((a.*b.*) |(b.*a.*))?	If a happens then b has to happen or happened after after a, and vice versa.
Simple ordered	Response(a,b)	$\Box(a \implies \Diamond b)$	[^a]*(a.*b)*[^a]*	Whenever activity a happens, activity b has to happen eventually afterward.
	Precedence(a,b)	$(\neg b\, U\, a) \vee \Box(\neg b)$	[^b]*(a.*b)*[^b]*	Whenever activity b happens, activity a has to have happened before it.
	Succession(a,b)	$response(a,b) \wedge precedence(a,b)$	[^ab]*(a.*b)*[^ab]*	Both Response(a,b) and Precedence(a,b) hold.
Alternating ordered	Alternate response(a,b)	$\Box(a \implies \bigcirc(\neg a\, U\, b))$	[^a]*(a[^a]*b[^a]*)*	After each activity a, at least one activity b is executed. A following activity a can be executed again only after the first occurrence of activity b.
	Alternate precedence(a,b)	$precedence(a,b) \wedge \Box(b \implies \bigcirc(precedence(a,b)))$	[^b]*(a[^b]*b[^b]*)*	Before each activity b, at least one activity a is executed. A following activity b can be executed again only after the first next occurrence of activity a.
	Alternate succession(a,b)	$altresponse(a,b) \wedge precedence(a,b)$	[^ab]*(a[^ab]*b[^ab]*)*	Both alternative response(a,b) and alternate precedence(a,b) hold.
Chain ordered	Chain response(a,b)	$\Box(a \implies \bigcirc b)$	[^a]*(ab[^a]*)*	Every time activity a happens, it must be directly followed by activity b (activity b can also follow other activities).
	Chain precedence(a,b)	$\Box(\bigcirc b \implies a)$	[^b]*(ab[^b]*)*	Every time activity b happens, it must be directly preceded by activity a (activity a can also precede other activities).
	Chain succession(a,b)	$\Box(a \Longleftrightarrow \bigcirc b)$	[^ab]*(ab[^ab]*)*	Activities a and b can only happen directly following each other.
Negative	Not co-existence(a,b)	$\neg(\Diamond a \wedge \Diamond b)$	[^ab]*((a[^b]*) |(b[^a]*))?	Either activity a or b can happen, but not both.
	Not succession(a,b)	$\Box(a \implies \neg(\Diamond b))$	[^a]*(a[^b]*)*	Activity a cannot be followed by activity b, and activity b cannot be preceded by activity a.
	Not chain succession(a,b)	$\Box(a \implies \neg(\bigcirc b))$	[^a]*(a+[^ab][^a]*)*a*	Activities a and b can never directly follow each other.
Choice	Choice(a,b)	$\Diamond a \vee \Diamond b$	.*[ab].*	Activity a or activity b has to happen at least once, possibly both.
	Exclusive choice(a,b)	$(\Diamond a \vee \Diamond b) \wedge \neg(\Diamond a \wedge \Diamond b)$	([^b]*a[^b]*) |.*[ab].*([^a]*b[^a]*)	Activity a or activity b has to happen at least once, but not both.

Table 1: An overview of Declare constraint templates with their corresponding LTL formula and regular expression.

parison with the procedural process modeling paradigm was made in order to get a grasp on the benefits and downsides of using either approach, especially from the viewpoint of the reader and modeler. A case study with practitioners showed that, rather than using a full declarative approach, a hybrid approach suits interests best [26]. Since the control flow in declarative models is often underspecified, there is a vast number of execution scenarios which might impede the user's understanding of the actual behavior of a DPM. An investigation into the size and understandability of models was performed using the Alaska Simulator in [27], which presents a process as a journey and was used in a user study for solving planning problems. Later, a test suite for DPMs was introduced in [7, 28], looking into how users and modelers make use of the interplay and changes of constraints in a model. A comprehensive study of the understandability factors, the notation, usage of hierarchy [29], and interpretation strategies of users, was performed in [30]. In the latter works, it became apparent that *hidden dependencies* and the interplay of constraints clearly impede the understandability and raise the cognitive load of the reader and modeler. A hidden dependency in software was defined in [5, 31] as 'a relationship between two components such tat one of them is dependent on the other, but that the dependency is not fully visible'. They raise cognitive complexity [32] due to intertwining parts of software, and often are urged to be added visually to raise understandability [33].

2.3. Alleviating Understandability Issues: Discovering Hidden Dependencies

As will be illustrated below, hidden dependencies arise in DPMs when activities propagate their cardinalities, or are prevented from executing at a certain point in time, i.e., by *negative* constraints in Declare or *exclude* relations in DCR Graphs. In this paper, mitigation is sought for by making all dependencies between constraints that are not explicit in the model itself (i.e. hidden) visible in the form of textual annotations and dependency graphs. This approach is general to the extent that new constraints in any type of semantics can be added, as long as the basic principles of constraint-based models listed

are followed.

3. Preliminaries

In this section, the concept of a DPM and its constraints is formalized.

3.1. Constraint-Based Declarative Process Models

A declarative process model $DM = (A, \Pi)$ can be defined as follows.

- A is a set of activities or atomic propositions from the alphabet Σ,
- $\Pi(A)$ is a set of constraints defined over the activities,
- $\S(\pi)$, $\pi \in \Pi$ is a function assigning semantics to a constraint, and
- $\Phi = \bigwedge_{\pi \in \Pi} \S(\pi)$ is the model comprised of the conjunction of constraints, given that the language used to express the constraints is closed for common properties such as concatenation, intersection, Kleene star, and so on, as is the case for regular expressions and LTL.

For every activity $a \in A$ and timestamp $t \in \mathbb{N}$, we define

- $L : (a, t) \to \mathbb{N}$ the lower bound of the amount of occurrences of an activity at time t,
- $U : (a, t) \to \mathbb{N}$ the upper bound of the amount of occurrences of an activity at time t,
- $E : (a, t) \to \{0, 1\}$ a function keeping track of the enabledness of an activity at time t,
- $O : (a, t) \to \{0, 1\}$ a function indicating whether an activity fired at time t, and
- $\#_A(a, t) = \sum_{i=0}^{t} O(a, i)$ the number of times an activity has occurred up until time t.

For every constraint $\pi \in \Pi$ we can also define

– $C : (\pi, t) \to \{0, 1\}$, a function keeping track of the satisfaction status of the constraint at time t.

3.2. Declare Constraints and Their Characteristics

Declare models are typically constructed by using the set of constraints of Table 1. They range from unary constraints, indicating the position and cardinality of an activity, to n-ary constraints, which capture typical sequence behavior such as precedence and succession relationships. A Declare model is the conjunction of its constraints, $\phi_{DM} = \bigwedge_{\pi \in \Pi} \S_{Declare}(\pi)$, where $\S_{Declare}$ corresponds to the semantics available in Table 1. The constraints can be represented in more detail as follows:

– $\Pi_1(a, P)$ is the set of unary constraints with P its properties indicating cardinalities or position, and

– $\Pi_2(a, b)$ where $a, b \in A$ is the set of binary constraints that follow an activation/resolution strategy.

In literature, a and b are, depending on the constraint, typically referred to as antecedent and consequent [34, 35, 36]. In general a serves as the antecedent, except for *precedence* constraints, where this relation is reversed. For some constraints, both a and b serve as antecedent an consequent at the same time, i.e., for *(not) responded/co-existence*, *(exclusive) choice*, and *succession*. There also exist constraints that use a set for a or b called branched constraints [37], most notably the target/consequent-branched constraints [35]. In this case, multiple consequents can resolve the status of a constraint. For, e.g., *response(a,B)*, the LTL-formula becomes $\Box(a \implies \diamond(\bigvee_{b \in B} b))$ and any $b \in B$ can resolve the temporary violation caused an occurrence of a. These constraints are supported by the approach, but are not included. Only the example of *response(a,B)* will be elaborated.

Finally, for an activity $a \in A$ we define:

– $\bullet a = \{\pi \mid \pi(b, a) \in \Pi_2 \vee \pi(a, P) \in \Pi_1\}$ all prefix constraints (and unaries) of an activity, and

– $a\bullet = \{\pi \mid \pi(a, b) \in \Pi_2\}$ all postfix constraints of an activity.

Declare constraints exhibit a hierarchy, which is well-explained in [37] and [20]. For unary constraints, *existence(A,n)* and *absence(a,n)* together form *exactly(a,n)*. Binary constraints are divided in different classes, for which every class depends on the previous one: *Unordered (responded/co-existence)*, *Simple ordered (precedence (p), response (r), succession (s))*, *Alternating ordered (alternate p,r,s)*, and *Chain Ordered (chain p,r,s)*. Next to these constraints, there exist negative versions for three of them *(not co-existence, not succession, not chain succession)*. Finally, the *choice* constraint exists, which is comparable with a branched unary constraint *existence({a,b},1)*. For binary constraints, *(alternate/chain) response(a,b)* and *(alternate/ chain) precedence(a,b)* form *(alternate/chain) succession* respectively. When a property is discussed for, e.g., *precedence*, this hence also includes *(chain/alternate) succession* and *co-existence*. In the remainder of the text, the set of, e.g., *precedence* constraints is denoted as Π_{prec} with $\Pi_{chaiprec} \subseteq \Pi_{altprec} \subseteq \Pi_{prec}$.

3.3. Hidden Dependencies

The execution of a Declare model can be realized by constructing an automaton (either a Büchi [37] or finite state automaton (FSA) [20, 21]) by conjoining the different constraints' automata to obtain the behavior that is allowed for by all of them. This conjunction actually abolishes the notion of the separate constraints and thus throws away the information of how the separate constraints interact. One technique to mitigate this is to color the global automaton [19] by keeping both the global and separate automata, still the interactions are untraceable. *We define a hidden dependency as an interaction between constraints and their activities that is not made explicit as such in the model and its executable counterpart.* Each constraint has specific characteristics that are discussed in [38] that cause these dependencies:

– Some constraints have an impact on the temporary violation aspect of the model as they can be in a non-accepting state and require an activity

to resolve this temporary violation, i.e. *existence*, *response*, and *choice* constraints.

– Some constraints can disable activities for the remainder of the execution, i.e. the *absence*, *not succession*, *exclusive choice*, and *not co-existence* constraints.

– Some constraints can temporarily block all other activities, i.e. all *chain* constraints.

E.g., consider the model in Figure 1 consisting of $A = \{a, b, c\}$ and $\Pi = \{\pi_{resp}(a, b), \pi_{resp}(b, c), \pi_{exa}(c, 2)\}$. It contains a hidden dependency between *exactly(a,2)* and *response(a,b)*. When c is fired once (and hence can only fire one more time), and a has fired without b firing already, c should not fire before b resolves the temporary violation of $response(a, b)$. Since after firing c, c cannot resolve $response(b, c)$ anymore (as it can only fire two times) and b should not fire to avoid another temporary violation of $response(b, c)$.

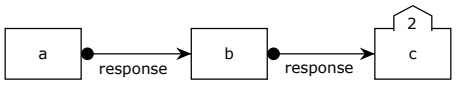

Figure 1: An example of a small Declare model with hidden dependencies.

4. Dependency Structures

This section discusses how dependency structures retrieved from constraint-based models can be constructed (Section 4.1) and how they can aid interpretation of the model regarding the way in which constraints interact (Section 4.2).

4.1. Construction

Hidden dependencies are caused by the implications that constraints pose on activities and the propagation of these implications through other constraints connected to that activity. To indicate these properties, the functions L and

U are used: its constraints can require the activity to either fire at least a number of times still (L), or to stop firing after a limited amount of times (U). An activity a is not enabled anymore when $U(a,t) = 0$, or $U(a,t) = 0 \implies E(a,t_l), \forall t_l > t$. In Table 2, an overview is given for all constraints how they propagate these bounds, as well as their impact on the enabledness on a certain time $E(a,t)$. They are derived directly from the definitions of the constraints. Again, all statements formulated for, e.g., *precedence(a,b)* also hold for *(chain/alternate) precedence/succession*. In case a declarative process model is not in a permanently violated state, $U(a,t) \geq L(a,t), \forall a,t$ holds as the lowerbound is always calculated as the maximum of the old and newly assigned value, and the upperbound is the minimum of the old and newly assigned value. E and C are determined by the outcome of the operationalization of the separate constraints, e.g. an FSA, and the outcome of applying the rules in the table iteratively over all activities. $E(A, t+1|\pi(a,b))$ is calculated as in Algorithm 1 where $\pi(a,b)$ is the constraint for which the rule is applied, i.e. *chain response* or *not chain succession*. The procedure checks whether the activity is enabled and whether it will become enabled through firing the antecedent of the constraint in question which may serve as a consequent in a *precedence* relationship.

For a branched *response(a,B)* constraint, the implications on the activity bounds become:

- $\Sigma_{b \in B} U(b,t) = 0 \implies U(a,t) = 0$,
- $\exists b \in B, U(b,t) = 1 \wedge \Sigma_{b_o \in B \setminus b} U(b_o,t) = 0$
  $\wedge L(a,t) > 0 \implies E(b,t+1) = 0$, and
- $(L(a,t) > 0 \vee C_A(t) = 1) \wedge \Sigma_{b_o \in B \setminus b} U(b_o,t) = 0 \implies L(b,t) = \max(L(b,t), 1)$.

The same exercise can be repeated for other branchable constraints as well. By propagating all these dependencies whenever a change is made to an activity until all bounds are set, the dependencies can be made explicit throughout the model. To explain how the constraints relate exactly, dependency structures are constructed to link parts of the model into constraint groups. Each constraint

Constraint	**Implications for Activity Bounds**	**Motivation**
Existence(a,n)	$L(a,t) = \max(n - \#_A(a,t), L(a,t))$	The lower bound of a is the maximum of either the amount a had to fire (n) minus the amount fired, and the lower bound as imposed through other constraints.
Absence(a,n)	$U(a,t) = \min(n - 1 - \#_A(a,t), U(a,t))$	The upper bound of a is the minimum of the amount of times a can still fire, and the upper bound as imposed through other constraints.
Responded existence(a,b)/ Precedence(b,a)	$L(a,t) > 0 \wedge C(t) = 0 \implies L(b,t) = \max(L(b,t), 1)$	If a still has to fire, b has to be able to resolve the constraint, unless this has already happened.
	$U(b,t) = 0 \wedge C(t) = 0 \implies U(a,t) = 0$	If b cannot fire anymore and the constraint is not activated, a cannot inflict a constraint and becomes disabled.
Response(a,b)	$U(b,t) = 0 \implies U(a,t) = 0$	If b cannot fire anymore, a cannot fire anymore as the violation cannot be resolved anymore.
	$U(b,t) = 1 \wedge L(a,t) > 0 \implies E(b,t+1) = 0$	If b can fire only once anymore, it cannot fire until a does not have to fire anymore.
	$L(a,t) > 0 \vee C(t) = 1 \implies L(b,t) = \max(L(b,t), 1)$	If the constraint is not satisfied or a still has to fire, b has to fire at least once still.
Alt. response(a,b)	$U(a,t) = \min(U(b,t) - C(t), U(a,t))$	a can only fire as many times as all $b \in B$ can still fire.
	$L(b,t) = \max(L(a,t) + C(t), L(b,t))$	The sum of the minimum occurrence of all $b \in B$ has to be able to cover the minimum number of times a still has to occur.
	$U(b,t) < L(a,t) \implies E(b,t+1) = 0$	If a still has to fire $L(a,t)$ times, b has to be able to accommodate that number. Once its upperbound falls below that number, it becomes disabled (temporarily).
Alt. precedence(b,a)	$U(a,t) = \min(U(b,t) + C(t), U(a,t))$	a can only fire as many times as $b \in B$ can still fire.
	$L(b,t) = \max(L(a,t) - C(t), L(b,t))$	The sum of the minimum occurrence of b has to be able to cover the minimum number of times a still has to occur.
	$U(a,t) < L(b,t) \implies E(a,t+1) = 0$	If b still has to fire $L(b,t)$ times, a has to be able to accommodate that number. Once its upperbound falls below that number, it becomes disabled (temporarily).
Chain response(a,b)	$E(b,t+1\|\pi(a,b)) = 0 \implies E(a,t) = 0$	If b cannot fire in the next position, a cannot fire to avoid irresolvable violation.
Chain precedence(a,b)	$LB(a) = \max(LB(a), LB(b))$	The lower bound of a should always be at least the lower bound of b, because the lower bound can only be lowered when b fires immediately after a.
Not succession(a,b)	$L(b,t) > 0 \implies E(a,t+1) = 0$	If a still has to fire, it cannot fire until b does not have to fire anymore.
Not co-existence(a,b)/ Exclusive choice(a,b)	$\#_A(a,t) > 0 \vee L(a,t) > 0 \implies E(b,t+1) = 0$	If a fired or still has to fire, b cannot fire anymore.
	$\#_A(b,t) > 0 \vee L(b,t) > 0 \implies E(a,t+1) = 0$	If b fired or still has to fire, a cannot fire anymore.
Not chain succession(a,b)	$\nexists c \in A \setminus \{a\}, E(c,t+1\|\pi(a,b)) = 1 \wedge \exists d \in A \setminus \{a\}, L(d,t) > 0 \implies E(a,t) = 0$	If there is no activity other than a that is enabled in the next position and there exists an activity that still has to fire, a cannot be enabled.
Choice(a,b)	$C(t) = 0 \wedge U(b,t) = 0 \implies L(a,t) = \max(L(a,t), 1)$	If b cannot fire anymore, a still has to happen at least once.
	$C(t) = 0 \wedge U(a,t) = 0 \implies L(b,t) = \max(L(b,t), 1)$	If a cannot fire anymore, b still has to happen at least once.

Table 2: Table explaining the relation of every constraint towards the bounds of its activities.

Algorithm 1 Calculating $E(c, t+1|\pi(a,b))$.

Output: $\{0, 1\}$

procedure $E(c, t+1|\pi(a,b))(c,\ \pi(a,b))$
  if $E(c,t) = 1$ **then** ▷ c is enabled
    for $\pi_i(a_i, b_i) \in {\bullet}c$ **do**
      if $\pi_i \in \Pi_{PD} \cup \Pi_{notchaisuc} \wedge a = a_i$ **then** ▷ Check whether c
        return 0 ▷ will not be disabled by a
  else
    for $\pi_i(a_i, b_i) \in {\bullet}c$ **do**
      if $\pi_i \in \Pi_{prec}$ **then** ▷ Only a *precedence* constraint can enable c
        if $C_A(\pi_i) = 0 \wedge a = a_i$ **then**
          return 0 ▷ c will not be enabled by a
  return 1

has different implications, and from Table 2, the following types are derived:

- **Backward-propagating constraints:** all *response*-type constraints require that the consequent resolves the temporary violation that might be triggered by the antecedent. Hence, all constraints that have the antecedent working as a consequent, being 'on the left hand side' directly influence the consequent, for it needs to resolve any outstanding temporary violations and hence has its lower bound raised. $\Pi_{BW} = \Pi_{respex} \cup \Pi_{coex} \cup \Pi_{resp/suc}$.
- **Forward-propagating constraints:** all *precedence*-type constraints require that the consequent fires to activate the antecedent. Hence, all constraints that require the antecedent to act as a consequent to resolve a violation rely on this type of constraints. $\Pi_{FW} = \Pi_{coex} \cup \Pi_{prec/suc}$.
- **Permanently-disabling constraints :** *not succession(a,b)*, *not chain succession(a,b)*, and *exclusive choice(a,b)* all permanently disable either both a and b, or only b, as the cumulative function $\#_A$ is used to set the bounds. Once the activities are disabled, they cannot become enabled again. The same holds for the *absence* and hence *exactly* constraint. $\Pi_{PD} = \Pi_{notsuc} \cup \Pi_{notcoex} \cup \Pi_{exclchoi}$.

Now we construct the set of dependency structures DP for DM with a dependency structure being a tuple $DS = (\pi^{DS}, \Pi^{DS}_{dep}, DS^{DS}_{dep}), DS \in DP$ with

- π^{DS} the constraint triggering the structure,

– Π_{dep}^{DS} the set of dependent constraints, and

– DS_{dep}^{DS} the set of nested dependency structures dependent of π^{DS}.

To fill Π_{dep}^{DS} and DS_{dep}^{DS}, Algorithm 2 creates a dependency structure for every activity that is involved in at least one of the five constraints that can permanently disable it. Hence, a structure is created for a in *absence/exactly(a,n)*, a and b in *exclusive choice/not co-existence(a,b)*, and for b in *not succession(a,b)* as can be seen in Algorithm 2, lines 7-25.

First, all backward-propagating constraints are considered ($\Pi_{BW} \subseteq \Pi$ and used for recursive search, as well as stored in Π_{dep} (Algorithm 3, lines 1-22) as they have a direct impact on π^{DS}. During this procedure, all incoming *existence* and *choice* constraints are stored as well (Algorithm 3, lines 16-18). They also need to be fulfilled, but do not propagate due to their unary nature. When *responded existence* is encountered, a new dependency structure $DL \in DS_{dep}^{DS}$ is constructed because when the constraint becomes satisfied (by firing its consequent), it is satisfied indefinitely (unlike, e.g., *response* which can become temporarily violated again) and its propagation is also abolished (Algorithm 3, lines 6-10).

For every activity that is encountered by the algorithm, a forward-dependency search is performed for all forward-propagating constraints $\Pi_{FW} \subseteq \Pi$, which includes all (*alternate/chain*) *precedence* constraints and *co-existence*. These constraints need to be activated (the antecedent has to be fired, in the case of alternating versions even multiple times) to resolve dependencies from backward-propagating constraints. The constraints dependent of them are linked to them through a separate, nested dependency structure $DL \in DS_{dep}^{DS}$ (Algorithm 3, lines 23-36). Since all constraints are considered in the bounds' knowledge base, all propagations, and hence dependencies, are calculated throughout the model. The dependency structures provide a visual summary of the constructs for which the calculations are impacting the behavior of the model on activities that are not local, i.e., directly connected to the other activities. The completeness of the technique actually stems from the bound propagations that also drive

Algorithm 2 Retrieving Dependency Structures

Input: $DM = (A, \Pi)$
Input: $\Pi_{BW} \leftarrow \Pi_{resp/coex} \cup \Pi_{resp}$ ▷ Backward-propagating constraints
Input: $\Pi_{FW} \leftarrow \Pi_{coex} \cup \Pi_{prec}$ ▷ Forward-propagating constraints
Output: DP ▷ The set of dependency structures for DM

procedure ReturnDepTrans(DM)
  $DP \leftarrow \emptyset$ ▷ The set of all dependency structures of the model
  for $\pi \in \Pi$ **do**
    $DS \leftarrow \emptyset$ ▷ The dependent structure for π
    $V^l \leftarrow \emptyset$ ▷ Set of visited activities for left search
    $V^r \leftarrow \emptyset$ ▷ Set of visited activities for right search
    if $\pi \in \Pi_{abs} \vee \pi \in \Pi_{exa}$ **then**
      $\pi^{DS} \leftarrow \pi,\ DS \leftarrow SeaLe(\pi_a, V^l, DS) \cup SeaRi(\pi_a, V^r, DS),\ DP \leftarrow DS$
    if $\pi \in \Pi_{notsuc}$ **then**
      $\pi^{DS} \leftarrow \pi,\ DS \leftarrow SeaLe(\pi_b, V^l, DS) \cup SeaRi(\pi_b, V^r, DS),\ DP \leftarrow DS$
    if $\pi \in \Pi_{exclchoi} \vee \pi \in \Pi_{notcoex}$ **then**
      $\pi^{DS} \leftarrow \pi,\ DS \leftarrow SeaLe(\pi_a, V^l, DS) \cup SeaRi(\pi_a, V^r, DS)$
      $DP \leftarrow DS,\ DS_2 \leftarrow \emptyset$
      $\pi^{DS_2} \leftarrow \pi,\ DS_2 \leftarrow SeaLe(\pi_b, V^l, DS) \cup SeaRi(\pi_b, V^r, DS_2)$
      $DP \leftarrow DS_2$
  return DP

the textual annotations.

4.2. Interpretation

All the updates that are performed throughout the model which change the upper and lower bounds of an activity because of unary propagation which are caused by other activities and constraints not directly connected to that activity, are externalizations of hidden dependencies. While the unary propagations provide the rationale for explaining hidden dependencies, dependency structures are used to visualize them. Although constructing dependency structures can already give extra information by displaying them in a graph showing which constraints interact with the main constraint (π^{DS}) in the structure, they can also be expressed in extra descriptions to annotate the model in order to help understand why constraints are related and what combined impact they have. These descriptions can be provided next to the model and are based on the following principles.

First of all, for *exclusive choice(a,b)* and *not co-existence(a,b)*, the structures reflect that whenever an activity from either structure is fired (either the one for a or b), the activities in the other structure become disabled permanently ($UB(a)/UB(b) = 0$). Indeed, firing any activity in the dependency structure

Algorithm 3 Search for Dependent Constraints

```
1: procedure SeaLe(a, V, DS)                                   ▷ Search the left hand side of the activity
2:     if ¬(a ∈ V) then                                        ▷ Do if a is not visited yet, avoids infinite loops
3:         V ← a
4:         for π ∈ •a do                                       ▷ Scan all incoming Declare constraints of activity a
5:             if π ∈ Π_BW then
6:                 if π ∈ Π_respex then
7:                     DL ← ∅                                  ▷ Create new nested dependency structure
8:                     π^DL ← π, DL ← SeaLe(π_a, V, DL) ∪ SeaRi(π_a, V, DL)
9:                     DS^DS_dep ← DL                          ▷ Add nested structure to main structure DS
10:                else
11:                    Π^DS_dep ← π, DS ← SeaLe(π_a, V, DS) ∪ SeaRi(π_a, V, DS)
12:            if π ∈ Π_exis ∨ π ∈ Π_exa ∨ π ∈ Π_choi then
13:                Π^DS_dep ← π
14:    return DS

15: procedure SeaRi(a, V, DS)                                  ▷ Search the right hand side of the activity
16:    if ¬(a ∈ V) then
17:        V ← a
18:        for π ∈ a• do                                       ▷ Scan all outgoing Declare constraints of activity a
19:            if π ∈ Π_FW then
20:                DL ← ∅, π^DL ← π
21:                DL ← SeaLe(π_b, V, DL) ∪ SeaRi(π_b, V, DL)
22:                DS^DS_dep ← DL
23:    return DS
```

of a or b requires a or b to fire, hence activating *exclusive choice* or *not co-existence*. If the structures of a and b share activities, this means the net is not deadlock-free.

Secondly, for *not succession(a,b)*, a becomes disabled whenever a constraint $\pi \in \Pi^{DS}_{dep}$ is temporarily violated and needs b to resolve it (i.e. $LB(b) > 0$). Also, dependent structures in $d \in DS^{DS}_{dep}$ cannot contain any violations in their Π^{d}_{dep} unless the antecedent of the main constraint $\pi^d \in DS^{d}_{dep}$ is activated and can execute a minimum number of times required.

For unary constraints, *absence(A,n)* and *exactly(A,n)*, this applies as well, with the exception that a becomes disabled when a constraint relies upon it to become satisfied again ($UB(a) = 1$ but there exist activities in Π^{DS}_{dep} for which the lower bound is higher than 0).

Finally, every execution of activities in *chain* constraints should be checked. For each of them, it is checked whether the consequent is available to fire for *chain response*, or is the only one available for *not chain succession* in order to avoid deadlock as in Algorithm 2.

4.3. Example

Consider the model in Figure 2. The lower and upper bounds and the enabledness of the activities are added for clarification. *Not succession(f,d)*, so any occurrence of f cannot be followed eventually by d, causes the algorithm to construct a dependency structure for d. Backward searching will yield no constraints, however, a forward search adds a new dependency structure for *alternate precedence(d,e)*. If d cannot fire anymore ($UB(d) = 0$ through *not succession(f,d)*), any activity that still relies on d to enable it will become permanently disabled as well, or have an upper bound of 1 when the *alternate precedence* is activated. A backward search for e adds *alternate response(b,e)* to the set of dependent constraints, next the operation continues until the following structure is constructed: $DS = \{\pi^{DS} = \pi_{notsuc}(c, b), \Pi^{DS}_{dep} = \emptyset, DS_{dep} = \{\pi^{DS} = \pi_{altprec}(d, e), \Pi^{DS}_{dep} = \{\pi_{altresp}(b, e), \pi_{altresp}(a, b)\}, DS_{dep} = \{\pi_{DS} = \pi_{altprec}(b, c), \Pi^{DS}_{dep} = \pi_{exis}(c, 2), DS_{dep} = \emptyset\}\}\}$. The dependency structure can also be visualized as in Figure 3. In this model, situations with very tricky implications can occur because of the interplay of constraints. As long as the lower bound of e does not reach 1 and *alternate precedence(d,e)* is not activated, f cannot fire. When c has fired once and a has fired, b needs to fire only once anymore to enable c for a second time. Hence e has to fire only once anymore afterwards to resolve *alternate response(b,e)*, and f can fire. If e would fire before reaching a lower bound of 1, d would have to fire in order to make e enabled again, hence disabling f. In a case where d would fire for the last time and e can fire only once anymore, e becomes disabled in case a fires, as this would require b to resolve *alternate response(a,b)* first and hence e to resolve *alternate response(b,e)* afterwards. Since e can only fire once anymore, it cannot use its last execution until all previous constraint violations have been resolved before it can resolve the violations directly connected to it.

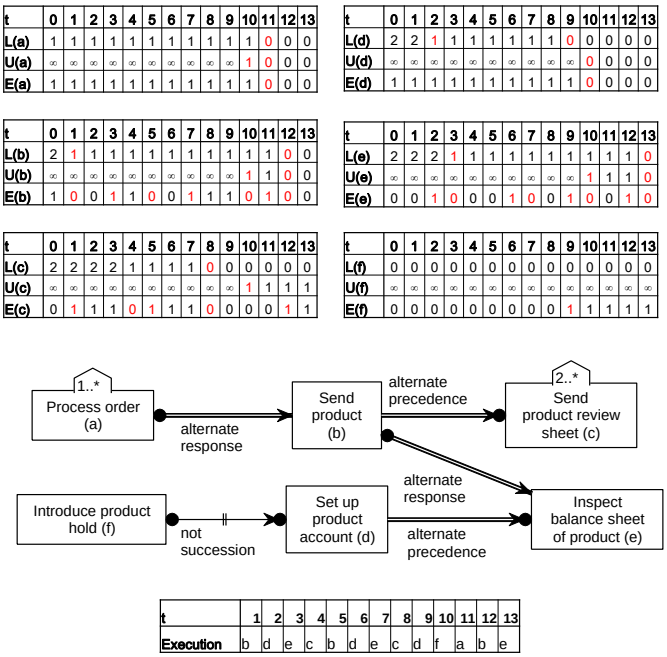

Figure 2: An example of a small Declare model with hidden dependencies with an example execution. The upper and lower bounds, as well as the enabledness, are visualized per activity.

5. Tool Support

The construction of the dependency structures has been implemented in the Declare Execution Environment, of which the implementation can be found by following the link in the introduction. The tool can read a Declare model saved from Declare Designer [39], which, during execution, is supported by descriptions for the hidden dependencies. A screenshot and an example can be found in Figure 4. Furthermore, the dependency structures can be visualized next to the model as a directed graph as well. Finally, the trace created over the model by the user is displayed below the model, aiding the user in understanding the history of the current situation displayed over the model. The execution semantics are provided by dk.brics.automaton [40] and consist of the product of the separate Declare automata expressed in regular expressions, as can be

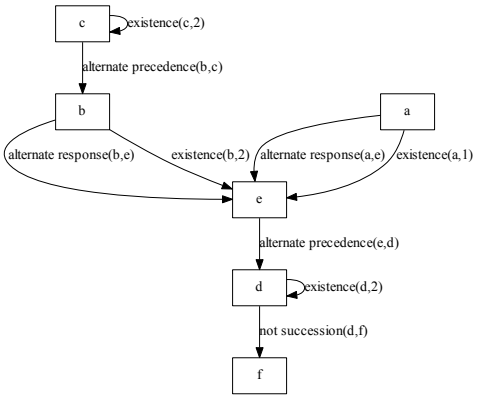

Figure 3: The corresponding dependency graph of the model in Figure 2.

found in [20] and [21].

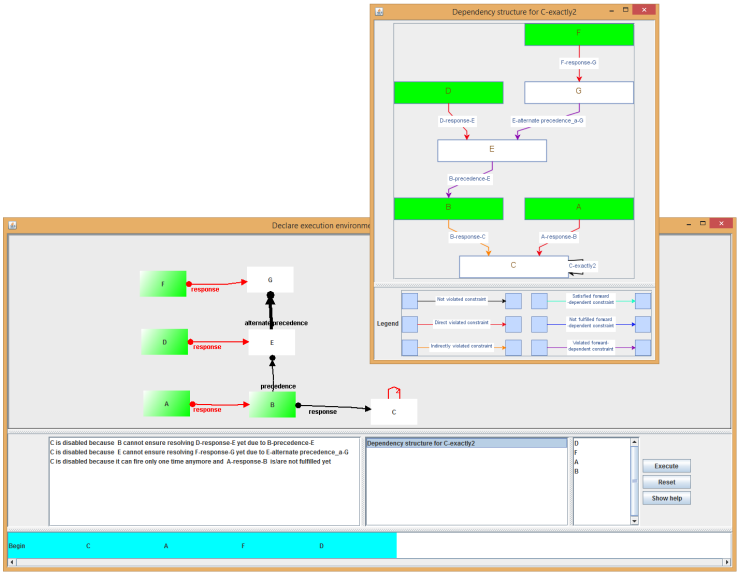

Figure 4: An example of a small Declare model with hidden dependencies and the corresponding dependency graph for *exactly(c,2)*. A high resolution version can be found by following the link to the tool's website.

6. Empirical Evaluation

In this section, it is empirically investigated whether the understandability issues and extra cognitive effort that hidden dependencies cause in declarative process models can be relieved by making them explicit.

6.1. Hypotheses

The presence of hidden dependencies has been shown to cause considerable impact on users' understanding of a DPM's behavior [41, 7]. Furthermore, it can be argued that hidden dependencies increase the cognitive effort needed to understand the model as they raise complexity by introducing extra relations between model constructs [42, 32]. Especially since negative constraints cause the hidden dependencies, they pose a significant threat to mental effort needed to resolve them, as humans typically do not think of what is not supposed to happen [43]. Hence, introducing textual annotations and dependency structures can reduce the extraneous cognitive load [44]. This has been proven successful in, e.g., the context of conceptual models by introducing subsystems and subject areas [45]. It will be measured whether there is any form of reduction of mental effort in terms of extraneous load, as dependency structures provide a different way of representing the information in declarative process models and hence might introduce computational offloading, and at least re-representation and graphical constraining [46]. The difference in extraneous load is tested by measuring the resulting score of answering questions regarding models containing hidden dependencies. Furthermore, it will be tested whether the mental effort is reduced by measuring the difficulty perceived by the users. The influence of the introduction of the separate constructs proposed earlier will be tested as well. Therefore, the following two null hypotheses are proposed:

H_0^1: The introduction of extra layers of annotation in declarative process models does not influence

a the understandability, as measured in the resulting score of solving exercises, and

b the mental effort, as measured in the perceived difficulty by the model's user, and as the time needed to solve the questions.

H_0^2: Extra visual information regarding hidden dependencies in the form of dependency structures has no impact on understandability, as measured in the resulting score of solving exercises.

6.2. Participants and Experimental Setup

In the experiment, 146 students participated. They were enrolled in KU Leuven's *Business Analysis* course, which is part of the Master's program in Business Engineering and includes material on both procedural and declarative process modeling. They were asked to solve five questions for each of five different Declare models in a timespan of two hours. The students, being from the same program with a similar educational background (Bachelor's in Business Engineering), were assumed to have the same modeling experience and and can be considered novice business process modelers. This was also tested with a question during the sessions, for which no-one indicated to have more than basic knowledge on BPMN, Petri nets, or Declare. The models are summarized in Table 3 and detailed descriptions, as well as graphical representations can be found on the tool's website. They were chosen in order to have a good representation of the different constraints that can cause hidden dependencies, and their interaction, i.e., they are present in multiple models, and also in different combinations. These models are used as the *model* variable in the statistical models in Section 6.3.1. The questions intently included overlaps to see whether participants progressed and learned to recognize the hidden dependencies when viewed from different angles. E.g., in model 5, all the different types of dependencies were brought together (in separate questions).

At the start of the test, students were provided with instructions on how to use the execution environment by making use of the example presented in Section 3.2, a model which was used as a foundation for models 3-5, but without the additional constraints and activities added. As such, the concept of

Model 1	**Model 2**	**Model 3**
Response(a,b)	Exactly(a,2)	Response(a,b)
Precedence(b,c)	Existence(c,2)	Response(b,c)
Not co-existence(b,e)	**Exactly(b,2)**	**Exactly(c,2)**
Response(d,e)	**Absence(d,3)**	Precedence(b,e)
	Alternate precedence(a,c)	Response(d,e)
	Alternate response(b,d)	Alternate precedence(e,g)
		Response(f,g)
Model 4	**Model 5**	
Response(a,b)	Response(a,b)	Choice(a,j)
Existence(b,1)	Response(b,c)	**Not succession(i,j)**
Alternate precedence(b,c)	**Exactly(c,2)**	
Not succession(b,e)	Precedence(b,e)	
Existence(c,1)	Response(d,e)	
Response(d,c)	Alternate precedence(e,g)	
Existence(d,2)	Response(f,g)	

Table 3: The different Declare models used during the experiments. The figures and detailed explanation of the models' behavior used in the experiment can be found by following the link to the tool site. The constraints causing hidden dependencies are indicated in bold.

constraint interrelations and hidden dependencies was explained but not introduced explicitly as the subject of study. Furthermore, the introduction using this model served as a tutorial for the participants to use the tool they were provided with.

In order to measure the impact of handing natural language descriptions and the visualization of dependency graphs, the students were divided into three groups (A (Female 18/Male 36), B (20/29), C (18/25)) which received a different version of the Declare Execution Environment. Group A could only see the Declare model and the constraint descriptions, but no color annotation nor dependency structure visualizations. Group B received a tool in which the enabled activities were colored green, and temporarily violated constraints were colored red, in a fashion described in [19] and similar to Declare Designer [39]. Since the colors are used for different constructs, only the shift of color had an importance, not the type of color used itself. Also, the constraint descriptions were given. Finally, group C was given an environment with the same functionality as group B, but with extra descriptions concerning hidden dependencies, as well as the possibility to open a dynamic visualization of the dependency structures. These groups are represented in the results by the *tool* variable.

Throughout the session, it was recorded how long each student needed so solve a particular question (measured in seconds), as well as whether they opened the dependency structures' visualization when being part of group C. After solving the questions of a model, students were asked to rate the difficulty of the model on a 5-point scale (from 'Very easy' to 'Very hard'). This was used to measure the perceived difficulty, as self-rating can be considered as a measure for mental effort [47, 30].

The questions were aimed at uncovering to which extent the participants grasped the full impact of the blend of different constraints. They were asked to indicate which activities were enabled after firing a certain sequence, and why or how to reach a certain firing sequence. Since two out of the three groups knew which ones were enabled, they could focus more on the second part of the question. An example question used for model 1 is 'After firing d, which activities are still enabled? Explain.'.

Each question was scored on a 0 to 1 scale, where incomplete answers (usually because of overlooked hidden dependencies or incorrect use of constraints) were still awarded a score higher than 0. E.g., a student from group B who provides the correct set of enabled activities but fails to state that activity c in model 3 is not enabled because of hidden dependencies was still awarded 0.6. The explanation was taken into account so as to make a fair comparison with students in group A, who got no extra information, and therefore many times missed even these basic answers. Group C students that just copied extra descriptions provided by the tool also did no receive a grade of 1, as they did not prove to understand the model. The scores were awarded by one author, and were systematically checked by the other authors for consistency, following common correction criteria. To illustrate the correction procedure, consider the following three answers from students in group B for question 1 for model 3 'After firing the sequence A - C, which activities are enabled? Explain.'

– **Score 1.0:** 'A, B, F and D are enabled. C can not be executed because of the response relation between A and B and B and C. Given that we

already fired A, we know that eventually we'll need to fire B and also C. C can be fired only twice and since it has already been fired, we can not fire it again before firing B. G is not enabled because it can not happen before E and E hasn't been executed yet. E is not enabled due to the precedence relation between B and E and the fact that B hasn't been fired yet.'

- **Score 0.6:** 'A, B, D and F are enabled. If A fires, B has to be executed eventually afterwards. C can be executed exactly 2 times: since it has already fired + it will need to fire again when B will eventually fire, it is not enabled anymore. E is not enabled because it needs to be preceeded by B. B is not yet fired, so E is not enabled. A, D and F can still fire.' - It is not explained why C is not enabled, but at least the answer is correct.

- **Score 0.4:** 'A,B,D,C. E should be preceded by B, so E is not enabled. We cannot enable activity G because first activity E should happen. This is also the reason we cannot enable activity F, because after F has to follow G and G cannot happen before E.' - This answer fails to capture many relations in the model and incorrectly labels C as enabled.

6.3. Results

In this section, the results of the experiments are discussed.

6.3.1. Quantitative Results

Given this setup, an experimental analysis was conducted to investigate the impact of the execution environment students were given (i.e. tool) on the score with a higher score indicating a better level of understanding.

The scores are skewed to the right, as once a student completely grasped the full behavior of the model, her/his score reached 1 more easily over all questions and models. The Shapiro-Wilk normality test [48] rejects that the *score* follows a normal distribution with a p-value smaller than 0.001. Hence, for statistical analysis, non-parametric tests are advised. However, since the sample size is large and ANOVA can be considered robust to non-normality [49], multi-way

ANOVA was used with *score* as the dependent variable, and *model* and *tool* as independent variables. To ensure the correctness of the results, the aligned-rank transformation approach for non-parametric ANOVA [50], a good alternative for the Kruskal-Wallis and Friedman tests, was used to verify that no significant differences were present in the results when using non-parametric tests.

Next to the session and tool variables, it was tested whether gender had any explanatory power by performing an ANOVA analysis for $score = session + model + gender$ and its interaction effects. The main effects with the variables were not significant on a p-threshold of 0.05, however, some interaction effects were significant. Judging from a full regression performed on all variables and their interactions on the whole dataset, however, it was concluded that the effect sizes can be considered very small.

In Table 4, a pairwise mean comparison using Tukey's HSD tests is reported for *score*. Clearly, there is a big difference between the usage of the different tools, offering proof to reject hypothesis $\mathbf{H_0^1}$ that the annotations have no impact on the way the participants score for the questions. When comparing the models, the results also show very strong differences in the estimated values, except for the difference between models 3-4 on a 0.05 significance level. So as to evaluate the impact of each variable, a linear regression

	Difference	Lower bound	Upper bound	P-value
B-A	0.150	0.123	0.178	<0.001
C-A	0.262	0.234	0.290	<0.001
C-B	0.112	0.083	0.141	<0.001
2-1	-0.144	-0.186	-0.102	<0.001
3-1	-0.094	-0.136	-0.052	<0.001
4-1	-0.089	-0.131	-0.047	<0.001
5-1	-0.195	-0.238	-0.153	<0.001
3-2	0.050	0.008	0.092	0.011
4-2	0.055	0.013	0.097	0.004
5-2	-0.051	-0.093	-0.009	0.008
4-3	0.005	-0.037	0.047	0.998
5-3	-0.101	-0.143	-0.059	<0.001
5-4	-0.106	-0.148	-0.064	<0.001

Table 4: Tukey's HSD mean comparison for the scores for the *tool* (A-C) and *model* (1-5) variables.

$(score = \alpha \times model + \beta \times session + \gamma \times session \times tool + \epsilon)$ was fitted on the

data. From the results in Table 5, it is clear that the impact of both the model as well as the tool is highly significant. We have done post-tests to check the assumptions for linear regression, including running a Durbin-Watson-test [51] which rejected the hypothesis for correlation among the residuals. Finally, it was tested whether the error terms were distributed normally. From the effect sizes, it becomes clear that the impact on the scores is considerable. Students using tools B and C achieved significantly better scores, especially for model 5, the most difficult model, the students were better capable of answering the questions correctly.

Coefficients	**Estimate**	**Std. Error**	**t value**	**Pr(> |t|)**	
(Intercept)	0.662	0.018	36.739	<0.001	***
toolB	0.082	0.026	3.157	0.002	**
toolC	0.151	0.027	5.582	<0.001	***
model2	-0.212	0.025	-8.334	<0.001	***
model3	-0.152	0.025	-5.962	<0.001	***
model4	-0.130	0.025	-5.103	<0.001	***
model5	-0.307	0.025	-12.045	<0.001	***
toolB×model2	0.038	0.037	1.036	0.300	
toolB×model3	0.085	0.037	2.310	0.021	*
toolB×model4	0.062	0.037	1.673	0.094	.
toolB×model5	0.153	0.037	4.161	<0.001	***
toolC×model2	0.187	0.038	4.894	<0.001	***
toolC×model3	0.098	0.038	2.571	0.010	*
toolC×model4	0.067	0.038	1.761	0.078	.
toolC×model5	0.202	0.038	5.297	<0.001	***
Residual standard error: 0.295 on 3633 degrees of freedom					
Multiple R-squared: 0.163, **Adjusted R-squared:** 0.160					
F-statistic: 50.68 on 14 and 3633 DF, **p-value:** <0.001					

Table 5: Linear regression model for *score* based on the data gathered from the experiment with significance scores '***' 0, '**' 0.001, and '*' 0.01.

The average duration the students attributed to solving the questions, is displayed in Figure 5. The most notable difference, as was verified with an M-ANOVA, is the smaller amount of time spent on models 4 and 5 by students using tool A. It appears that the students became fatigued by the more challenging nature of the models, as is also clear from the scores. Especially model 5 seems to have been solved more quickly, contrary to its level of difficulty. In general, model 1 seemed to be easily tackled in a smaller timespan, compared to models 2-4. Participants in groups B and C did not record longer significantly different durations, which indicates that they probably had a sufficient amount

of time, until they ran out of the time near the end when answering questions for model 5.

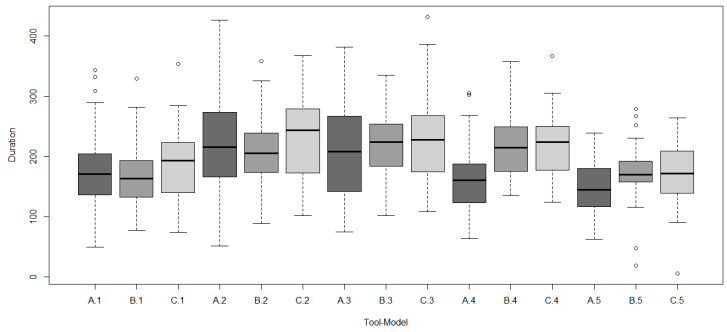

Figure 5: Boxplot of the average duration (in seconds) needed to solve the questions for the different tools (A-C) and models (1-5).

The perceived difficulty as indicated by the users is shown in Figure 6. There are notable differences, i.e., the most significant one being that users of tool C clearly found the models easier to understand in general, as the ratio of models perceived 'Very easy' and 'Easy' is higher. The users of Tool B tended to find more models 'Hard' to understand.

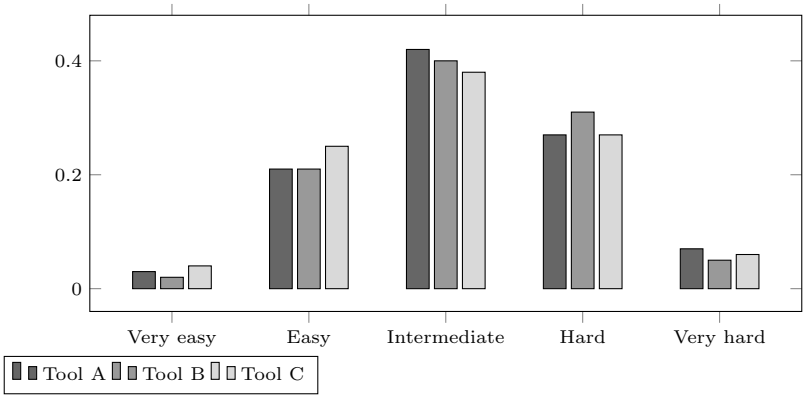

Figure 6: Percentage of perceived difficulty scores of the models per tool relative to the number of participants in the group.

For the participants using tool C, who were able to use the visualization

of the dependency structures, it was tested whether the ones who opened the visualization achieved higher scores with a non-parametric Mann-Whitney test. With a p-value of 0.693, it was not possible to find a significant difference, hence, hypothesis two cannot be rejected. Since the score of participants in this group is high, it might be that there was little room for improvement, or it is also possible that the textual annotations were sufficient in providing better understanding of the models' behavior. The results can be found on the tool's site.

In conclusion, we find that there is significant evidence to reject hypothesis $\mathbf{H_0^1a}$, i.e., the introduction of textual annotations improves the understandability, and there is also evidence suggesting that there is an impact on both the perceived difficulty and the duration needed to solve the questions, hence influencing mental effort ($\mathbf{H_0^1b}$). There is no evidence to reject $\mathbf{H_0^2}$, suggesting that the visualization of dependency structures does not improve the score of the model user when textual annotations are present as well.

6.3.2. Qualitative Results

Since the participants did not just give an answer in the form of 'a is now enabled' but had to motivate their answers, some extra observations can be made concerning the results. Although it was the case that the two groups with the more elaborate tool were better capable of seeing which activities are enabled and which constraints are violated, they still seemed to ignore these annotations. Especially group B sometimes (roughly 20%) ignored the coloring of the model as they did not understand some implications of the constraints. Participants often (30%) also bended the descriptions of the Declare constraints towards their understanding, hence starting to discuss irrelevant parts of the model. For the third group, this behavior was still present, although to a much lesser extent. Group A participants often found the hidden dependencies in the easier examples (up to 40% for question 3, only 10$ for question 5), because they had no support they thought harder about the model, but failed to find any in the elaborate examples. Furthermore, it was obvious from results that

any type of *response* constraint was very hard to grasp for students in general. Especially the constraint description that was adopted from the Declare tool [39] ('Whenever a happens, there needs to be a b...') seemed to cause confusion when used in a different context than in class. Some students started to interchange *response* and *alternate response* because of the 'Whenever A...' part.

6.3.3. Remarks

As in all empirical experiments, there are threats to validity that need to be addressed, most notably:

- **Construct validity:** Our experiment was threatened by the hypothesis guessing threat because students might figure out what the purpose of the study is, which could affect their guesses. We minimized this threat by hiding the goal of the experiment. Furthermore, it was only possible to test 5 models, which does not cover all the different dependencies. However, all the constraints that cause hidden dependencies were used in the questions to cover them at least in one model. Because the questions per model were complementary rather than independent, they are not reported separately, but aggregated. However, this might affect the comparison of certain questions that were easier to answer by using dependency structures and to compare the groups along this dimension.

- **Internal validity:** Our experiment had the maturation threat because subjects may react differently as time passes (because of boredom or fatigue). We solved this threat by dividing the experiment into different questions per model. Also, we made sure there could be no interaction between the students of different sessions. Also, the questions were always asked in the same order, mainly due to technical restrictions, and also to capture whether the participants progressed and learned from the previous examples. Not only over exercises, but within an exercise series the questions built upon each other to some extent. However, students received the base concept of how they needed to interpret a model with hidden de-

pendencies during the introduction, giving them a clue for models 1 and 3-5.

– **External validity:** Our experiment might suffer from interaction of selection and treatment: the subject population is limited to students. Although the number of subjects is quite high and their profiles quite balanced, we can only generalize the results to students, but the subjects might not be representative to generalize the results to professional modelers as well. It is, e.g., not possible to claim that the tool can help or improve Declare modeling efforts of more experienced users.

Furthermore, it must be noted that larger hidden dependency structures can, similar to larger process models, be more difficult to comprehend. While the models used in the experiments are relatively small, the hidden dependency structures in model 5 were already of considerable size. Given that the scores are lower, it is worthwhile to investigate how informative larger hidden dependency structures still are in future research.

7. Conclusion and Future Work

This paper proposed and validated a new technique to raise the understandability of declarative process models by constructing explicit dependency structures between the model's activities and constraints. It offers a theoretic aspect in explaining how to construct and interpret the relations of constraints and their hidden dependencies in ways that have not been proposed yet, and a practical aspect in the tool and user experiment which demonstrated that explaining and visualizing hidden dependencies and constraint structures rendered users significantly better capable of understanding the models they had to interpret. In this respect, the paper contributes to both the formal literature on declarative process modeling, as well as to the body of research concerning the cognitive aspects of process model comprehension.

Future work may be summarized as follows. The notion of dependency structures can be used to construct a cognitive complexity metric that is tai-

lored towards constraint-based process models. Next, the expressions used for calculating the propagation of lower and upper bounds of activities can be applied towards model checking and verification. By applying the rules, mistakes in process models can be easily identified and traced back to the constraints that are actually involved.

References

[1] M. Pesic, W. M. P. van der Aalst, A declarative approach for flexible business processes management, in: Business Process Management Workshops, Vol. 4103 of Lecture Notes in Computer Science, Springer, 2006, pp. 169–180.

[2] S. Goedertier, J. Vanthienen, Declarative process modeling with business vocabulary and business rules, in: OTM Workshops (1), Vol. 4805 of Lecture Notes in Computer Science, Springer, 2007, pp. 603–612.

[3] M. Reichert, What BPM technology can do for healthcare process support, in: AIME, Vol. 6747 of Lecture Notes in Computer Science, Springer, 2011, pp. 2–13.

[4] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, S. Storari, Declarative specification and verification of service choreographiess, TWEB 4 (1).

[5] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, R. M. Young, Cognitive dimensions of notations: Design tools for cognitive technology, in: Cognitive Technology, Vol. 2117 of Lecture Notes in Computer Science, Springer, 2001, pp. 325–341.

[6] C. Haisjackl, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, B. Weber, Making sense of declarative process models: Common strategies and typical pitfalls, in: BMMDS/EMMSAD, Vol. 147 of Lecture Notes in Business Information Processing, Springer, 2013, pp. 2–17.

[7] S. Zugal, J. Pinggera, B. Weber, The impact of testcases on the maintainability of declarative process models, in: BMMDS/EMMSAD, Vol. 81 of Lecture Notes in Business Information Processing, Springer, 2011, pp. 163–177.

[8] D. Fahland, D. Lübke, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: The issue of understandability, in: BMMDS/EMMSAD, Vol. 29 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 353–366.

[9] J. D. Smedt, J. D. Weerdt, E. Serral, J. Vanthienen, Improving understandability of declarative process models by revealing hidden dependencies, in: Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings, 2016, pp. 83–98.

[10] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, DECLARE: full support for loosely-structured processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.

[11] W. M. P. van der Aalst, M. Pesic, Decserflow: Towards a truly declarative service flow language, in: The Role of Business Processes in Service Oriented Architectures, Vol. 06291 of Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[12] O. M. G. OMG Specification, Semantics of Business Vocabulary and Rules, accessed on 09.11.2016 (2015).

[13] T. T. Hildebrandt, R. R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, in: PLACES, Vol. 69 of EPTCS, 2010, pp. 59–73.

[14] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, Nested dynamic condi-

tion response graphs, in: FSEN, Vol. 7141 of Lecture Notes in Computer Science, Springer, 2011, pp. 343–350.

[15] M. Montali, F. Chesani, P. Mello, F. M. Maggi, Towards data-aware constraints in declare, in: SAC, ACM, 2013, pp. 1391–1396.

[16] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. F. T. H. III, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, R. Vaculín, Introducing the guard-stage-milestone approach for specifying business entity lifecycles, in: WS-FM, Vol. 6551 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–24.

[17] F. M. Maggi, A. J. Mooij, W. M. P. van der Aalst, User-guided discovery of declarative process models, in: CIDM, IEEE, 2011, pp. 192–199.

[18] M. L. Bernardi, M. Cimitile, G. A. D. Lucca, F. M. Maggi, Using declarative workflow languages to develop process-centric web applications, in: EDOC Workshops, IEEE Computer Society, 2012, pp. 56–65.

[19] F. M. Maggi, M. Montali, M. Westergaard, W. M. P. van der Aalst, Monitoring business constraints with linear temporal logic: An approach based on colored automata, in: BPM, Vol. 6896 of Lecture Notes in Computer Science, Springer, 2011, pp. 132–147.

[20] C. D. Ciccio, M. Mecella, A two-step fast algorithm for the automated discovery of declarative workflows, in: IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2013, Singapore, 16-19 April, 2013, 2013, pp. 135–142.

[21] M. Westergaard, C. Stahl, H. A. Reijers, UnconstrainedMiner: Efficient Discovery of Generalized Declarative Process Models, Tech. rep., Technische Universiteit Eindhoven (2013).

[22] J. D. Smedt, S. K. L. M. vanden Broucke, J. D. Weerdt, J. Vanthienen, A full R/I-net construct lexicon for declare constraints, Tech. rep., KU Leuven (2015).

[23] M. Chinosi, A. Trombetta, BPMN: an introduction to the standard, Computer Standards & Interfaces 34 (1) (2012) 124–134.

[24] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[25] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, H. A. Reijers, Imperative versus declarative process modeling languages: An empirical investigation, in: Business Process Management Workshops (1), Vol. 99 of Lecture Notes in Business Information Processing, Springer, 2011, pp. 383–394.

[26] H. A. Reijers, T. Slaats, C. Stahl, Declarative modeling-an academic dream or the future for bpm?, in: BPM, Vol. 8094 of Lecture Notes in Computer Science, Springer, 2013, pp. 307–322.

[27] B. Weber, S. W. Sadiq, M. Reichert, Beyond rigidity - dynamic process lifecycle support, Computer Science - R&D 23 (2) (2009) 47–65.

[28] S. Zugal, J. Pinggera, B. Weber, Toward enhanced life-cycle support for declarative processes, Journal of Software: Evolution and Process 24 (3) (2012) 285–302.

[29] S. Zugal, P. Soffer, J. Pinggera, B. Weber, Expressiveness and understandability considerations of hierarchy in declarative business process models, in: BMMDS/EMMSAD, Vol. 113 of Lecture Notes in Business Information Processing, Springer, 2012, pp. 167–181.

[30] C. Haisjackl, I. Barba, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, B. Weber, Understanding declare models: strategies, pitfalls, empirical results, Software and System Modeling 15 (2) (2016) 325–352.

[31] T. R. G. Green, M. Petre, Usability analysis of visual programming environments: A 'cognitive dimensions' framework, J. Vis. Lang. Comput. 7 (2) (1996) 131–174.

[32] S. N. Cant, D. R. Jeffery, B. Henderson-Sellers, A conceptual model of cognitive complexity of elements of the programming process, Information & Software Technology 37 (7) (1995) 351–362.

[33] N. Dulac, T. Viguier, N. G. Leveson, M. D. Storey, On the use of visualization in formal requirements specification, in: RE, IEEE Computer Society, 2002, pp. 71–80.

[34] A. Burattin, F. M. Maggi, W. M. P. van der Aalst, A. Sperduti, Techniques for a posteriori analysis of declarative processes, in: 16th IEEE International Enterprise Distributed Object Computing Conference 2012, Beijing, China, September 10-14, 2012, 2012, pp. 41–50.

[35] C. D. Ciccio, F. M. Maggi, J. Mendling, Efficient discovery of target-branched declare constraints, Inf. Syst. 56 (2016) 258–283.

[36] F. M. Maggi, R. P. J. C. Bose, W. M. P. van der Aalst, Efficient discovery of understandable declarative process models from event logs, in: CAiSE, Vol. 7328 of Lecture Notes in Computer Science, Springer, 2012, pp. 270–285.

[37] M. Pesic, Constraint-based workflow management systems: shifting control to users, Ph.D. thesis, Technische Universiteit Eindhoven (2008).

[38] J. D. Smedt, J. D. Weerdt, J. Vanthienen, G. Poels, Mixed-paradigm process modeling with intertwined state spaces, Business & Information Systems Engineering 58 (1) (2016) 19–29.

[39] M. Westergaard, F. M. Maggi, Declare: A tool suite for declarative workflow modeling and enactment, in: BPM (Demos), Vol. 820 of CEUR Workshop Proceedings, CEUR-WS.org, 2011.

[40] A. Møller, dk. brics. automaton–finite-state automata and regular expressions for Java, 2010, accessed on 09.11.2016.

[41] C. Haisjackl, S. Zugal, Investigating differences between graphical and textual declarative process models, in: CAiSE Workshops, Vol. 178 of Lecture Notes in Business Information Processing, Springer, 2014, pp. 194–206.

[42] K. Figl, R. Laue, Cognitive complexity in business process modeling, in: CAiSE, Vol. 6741 of Lecture Notes in Computer Science, Springer, 2011, pp. 452–466.

[43] F. Bodart, A. Patel, M. Sim, R. Weber, Should optional properties be used in conceptual modelling? A theory and three empirical tests, Information Systems Research 12 (4) (2001) 384–405.

[44] J. Sweller, Cognitive load during problem solving: Effects on learning, Cognitive Science 12 (2) (1988) 257–285.

[45] D. L. Moody, Cognitive load effects on end user understanding of conceptual models: An experimental analysis, in: ADBIS, Vol. 3255 of Lecture Notes in Computer Science, Springer, 2004, pp. 129–143.

[46] M. Scaife, Y. Rogers, External cognition: how do graphical representations work?, Int. J. Hum.-Comput. Stud. 45 (2) (1996) 185–213.

[47] R. E. Mayer, P. Chandler, When learning is just a click away: Does simple user interaction foster deeper understanding of multimedia messages?, Journal of educational psychology 93 (2) (2001) 390.

[48] J. P. Royston, An extension of Shapiro and Wilk's W test for normality to large samples, Applied Statistics (1982) 115–124.

[49] A. Khan, G. D. Rayner, Robustness to non-normality of common tests for the many-sample location problem, JAMDS 7 (4) (2003) 187–206.

[50] J. O. Wobbrock, L. Findlater, D. Gergle, J. J. Higgins, The aligned rank transform for nonparametric factorial analyses using only anova procedures, in: CHI, ACM, 2011, pp. 143–146.

[51] J. Durbin, G. S. Watson, Testing for serial correlation in least squares regression: I, Biometrika 37 (3/4) (1950) 409–428.